Assistance au test de modèles à composants et services

Pascal Andre, Gilles Ardourel, Jean-Marie Mottu

Assistance au test de modèles à composants et services

Pascal André, Gilles Ardourel and Jean-Marie Mottu

LINA UMR CNRS 6241
Université de Nantes
{Prénom.Nom}@univ-nantes.fr

Résumé

Dans l'ingénierie des modèles, la correction des modèles est essentielle. Tester le plus tôt possible permet de réduire le coût du processus de vérification et de validation. Distinguer modèle abstrait et modèle spécifique réduit la complexité du test et favorise l'évolution du système modélisé. Nous ciblons les modèles à composants et services, ayant un niveau de description suffisamment précis et détaillé pour pouvoir exécuter les tests. Notre objectif est de tester ces modèles à composants c'est-à-dire de concevoir des cas de tests, de les appliquer sur les modèles mis dans un contexte adéquat pour être exécutés et obtenir un verdict. Pour réduire l'effort de construction du harnais de test, nous proposons une méthode qui guide le testeur dans le processus de conception des tests au niveau du modèle. L'assistance à la construction est basée sur (1) la détection d'incohérences et d'incomplétude entre le harnais et le modèle de test ainsi que sur (2) des propositions générant les éléments manquants. Le programme de test est alors transformé vers une plateforme technique dédiée à l'exécution des tests. La mise en œuvre est réalisée avec des plugins Eclipse dans COSTO, une plateforme dédiée au modèle à composants Kmelia.

Mots-clés : Composant, Service, Correction, IDM, Harnais de test, Assistance

Abstract

In model-driven development, the correctness of models is essential. Testing as soon as possible reduces the cost of the verification and validation process. The abstract model and the concrete model are isolated to reduce the business test complexity and to improve the model evolution. We target model testing for Service-based Component Models. We assume a formal specification of the component models in which the level of detail is sufficient to run tests. Our goal is to test, diagnose in an suitable execution context and fix component models. To facilitate the test harness construction, we propose an approach to guide the testers through the process of designing tests at the model level. Our prototype detects mismatches between the component harness and the test description and facilities to solve them. The tests are then executed by transforming towards a technical platform dedicated to run the tests. This approach is illustrated with Kmelia, a service-based component specification language covering structural, functional and communication features.

Keywords: Component, Service, Correctness, MDD, Test Harness, Assistance

1 Introduction

En Ingénierie Dirigée par les Modèles (IDM), la correction des modèles est essentielle car ils sont le point de départ de transformations plus ou moins directes vers le modèle opérationnel et l'implantation. Nous traitons ici des modèles à composants et services, expressifs et couvrant la structure, la dynamique et le comportement fonctionnel du système. Assurer la correction de ces modèles reste un défi [12]. La vérification permet de filtrer une partie des modèles erronés mais elle ne suffit pas parce que les outils restent limités. Nous considérons le test pour améliorer le niveau de correction.

Tester le plus tôt possible permet de réduire le coût du processus de vérification et de validation [16]. Dans une vision pragmatique, l'objectif est de tester la correction directement sur les modèles [7]. Ainsi l'effort se concentre sur la détection en amont des erreurs "métiers" (*platform independent*), coûteuses à corriger lorsqu'elles sont repérées tardivement. Le test de modèles permet de s'affranchir des erreurs spécifiques à l'implantation et réduit la complexité globale du test [4]. Il permet une remontée des erreurs de test (*feedback*) en phase avec le langage de modélisation. Définir les tests au niveau modèle facilite leur adaptation en cas d'évolution du modèle. L'abstraction réduit la complexité du travail d'écriture des tests mais leur exécution sur des modèles reste problématique.

Pendant le test d'intégration de composants logiciels, l'encapsulation doit être préservée. Ainsi trouver où et comment fournir les données de test est difficile : pour atteindre une variable, il faut trouver les services qui la manipulent soit dans l'interface offerte du composant, soit dans des services qui en dépendent. Etablir le *contexte de chaque test* (*fixture*) d'un service impliquera une séquence d'invocation de services en plus de l'initialisation du composant. Dans [8], Gross mentionne trois défis pour le test de composants : (i) tester des composants dans un nouveau contexte [1], (ii) tester sans accéder au fonctionnement interne d'un composant, et (iii) déterminer le niveau d'acceptation des tests (adéquation). Ce dernier correspond au problème C1 dans la classification des problématiques de Ghosh et al. [6], qui comprend aussi la génération de données de test (C2), la sélection de sous-ensembles de composants à tester (C3) et la création de séquences de test de composants (C4). Dans cet article nous traitons les deux premiers défis de Gross et les problèmes C3 et C4 de Ghosh, qu'il faut résoudre avant d'envisager les autres défis et problèmes.

Nous considérons la construction de *harnais de test* unitaire ou d'intégration pour des systèmes à composants et services, qui permettent de passer des données de test, d'exécuter les tests, et de récupérer le verdict (succès ou échec du test). La construction part d'un modèle du système, d'une *intention de test* et d'informations devant être extraites des composants sans briser l'encapsulation. Sachant que le testeur découvre l'application à tester et que les intentions de test restent informelles, il a besoin d'itérer jusqu'à ce que le niveau de précision soit atteint pour que le système soit testable. Le testeur a besoin d'être assisté pour définir la forme véritable du test (où injecter les données de test, comment définir concrètement l'oracle) et déterminer la partie du système nécessaire au test.

Dans cet article nous proposons une approche d'assistance au test de conformité pour les modèles à composants et services. Cette approche intègre les tests au niveau modèle ; elle est outillée. Dans une *première contribution*, nous proposons de modéliser les tests au même niveau d'abstraction que les composants sous test. L'activité de test est intégrée au modèle à composant en ce sens que les tests deviennent eux-mêmes des composants modélisés et assemblés aux composants sous test. De ce fait, on bénéficie des outils de l'environnement de développement du modèle. Les composants de test peuvent eux-mêmes être transformés vers des plateformes spécifiques et l'effort de test en amont (*early testing*) est capitalisé. L'approche s'applique à différents phases du test, *e.g.* unitaire ou intégration, car le harnais de test peut contenir tout ou partie de l'application initiale. En ce sens, nous traitons les points de vue fournisseur et utilisateur discutés dans [8]. La *seconde contribution* est un

1. Ce peut être un nouveau développement par le fournisseur ou un déploiement par les utilisateurs.

processus guidé pour aider le testeur à gérer la complexité du harnais de test. Les tests sont orientés vers les services des interfaces de composants. Le harnais de test est obtenu à partir de transformations du composant sous test, guidés par l'objectif de test, enrichis par des composants de test, et assistés par plusieurs analyses heuristiques s'assurant de l'exactitude du modèle du harnais par le biais de vérifications. La *troisième contribution* est un outillage qui permet d'expérimenter l'approche. Il est mis en œuvre dans COSTO (*COmponent Study Toolbox*), la plate-forme Eclipse dédiée au modèle à composants Kmelia [1, 12]. Nous avons développé un *framework* en Java qui donne un cadre d'exécution et d'animation pour les modèles de test (plateforme spécifique). L'approche est illustrée sur un cas d'étude simple, un système de *platoon* de véhicules. La création et la qualification des cas de test ne sont pas traitées ici [2].

L'article est organisé de la façon suivante. Nous discutons dans la section 2 la problématique du test de modèle à travers un exemple. Le processus de construction du harnais de test est exposé dans la section 3. La section 4 décrit une mise en œuvre qui instrumente la proposition et l'illustre par une expérimentation. Nous détaillons des travaux complémentaires dans la section 5. La section 6 résume la contribution et trace des perspectives.

2 Tester des modèles à composants et services

Cette section introduit le test de modèles à composants et services ainsi que les problèmes rencontrés. Elle décrit notre approche via un exemple.

2.1 Modèles à composants et services

Un modèle de système est défini par des composants logiciels et des services. L'interface d'un composant précise les services qu'il offre et ceux qu'il requiert. L'interface d'un service fait de même et définit un contrat de service. Les services échangent des données par des canaux de communication qui réalisent les liens d'assemblage. On appelle *dépendance de service* l'ensemble des services invocables directement ou indirectement par un service. Les besoins d'un service sont (potentiellement) satisfaits en interne par d'autres services du même composant ou en externe, lorsque celui-ci est lié à un service offert d'un autre composant. Un *composite* est un composant qui encapsule un assemblage.

Dans l'exemple de la Figure 1, le composant c:Client requiert un service offert du composant s:Serveur, via un lien d'assemblage. Cet assemblage de composants est représenté selon la notation de SCA [11]. Nous avons étendu la notation SCA (partie droite de la légende

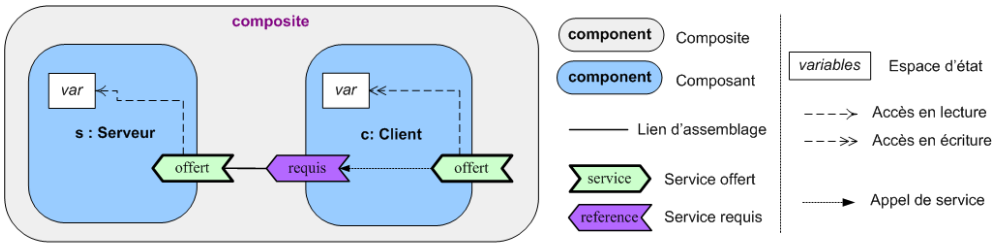


FIGURE 1 – Notation du modèle à composants et services

dans la Figure 1) pour mettre en évidence les dépendances de services et de données qui jouent un rôle prépondérant dans le contexte de test : (i) les composants ont un état formé de variables typées, (ii) les services ont un accès en lecture/écriture à ces variables, et (iii) les services offerts dépendent de services requis pour remplir leur tâche.

Illustrons ce modèle par l'exemple de *platoon* de véhicules de la Figure 2 [2]. Les véhicules, liés par ondes, adaptent leur état (vitesse et position) en fonction de leur prédécesseur, tout en préservant une distance de sécurité (propriété de sûreté de fonctionnement). Le pilote pilot est responsable du but. Trois (composants) véhicules sont enchaînés dans l'assemblage. Chaque composant dispose d'un service de configuration pour initialiser son état. Le service run place les véhicules dans un mode d'autorégulation qui requiert des services du véhicule précédent pour obtenir la position et la vitesse.

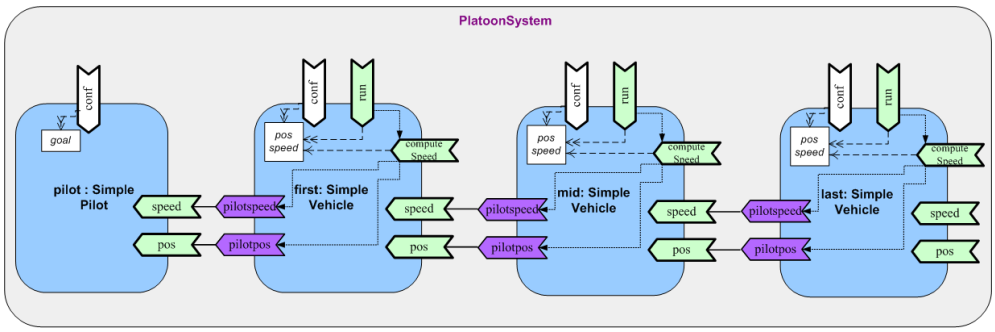


FIGURE 2 – Modèle à composants du système *Platoon* de véhicules

2.2 Test de conformité du modèle

Le *test de conformité* vise à montrer que les services des composants du système opèrent le comportement spécifié dans leur interface. Une partie du système, représenté par un modèle abstrait (PIM), est concernée par un test selon un objectif de test, parfois appelé intention de test. Ici le terme d'*intention de test* (IT) soutend l'objectif du test (en langage naturel), la cible (les services à tester) et les conditions de test (les données à fournir, l'état supposé des composants, les résultats attendus dans un oracle). Nous appelons *système sous test* (SUT - *system under test*), la partie du système concernée par le test.

Pour être testé, le SUT nécessite un environnement de test permettant de configurer les composants, passer des données de test, exécuter chaque test, en récupérer les sorties, contrôler leur conformité avec un oracle, produire leur verdict. Cet environnement est appelé un *harnais de test* (HT). Sa construction est une étape nécessaire avant de pouvoir considérer les techniques de création de données de test et d'oracle (par Model-based testing par exemple, ce qu'on ne traite pas ici). Des classes JUnit sont un exemple de harnais de test pour des programmes sous test écrits en Java.

Notre proposition consiste à modéliser le harnais au même niveau d'abstraction que le modèle à composant du système. Le processus que nous proposons (Figure 3) considère en entrée le modèle à composants du système que nous étudions au niveau modèle (Platform Independant Model), ainsi que l'intention de test. La première activité consiste à produire le harnais de test comme un *modèle spécifique au test* (Test Specific Model) qui va permettre de tester le SUT (un ou plusieurs composants sous test du système considéré). Ce niveau d'abstraction des tests implique de transformer le harnais vers une plateforme spécifique permettant l'exécution des tests et l'obtention de verdict. La deuxième activité du processus génère donc le code opérationnel des tests à partir d'une description du framework opérationnel (Platform Description Model).

Ce processus de test comprend deux étapes principales : la construction du harnais de test et sa mise en condition opérationnelle. Nous décrivons ces deux étapes dans la suite

2. L'exemple est disponible sur `http://www.lina.sciences.univ-nantes.fr/aelos/projects/kmelia/`

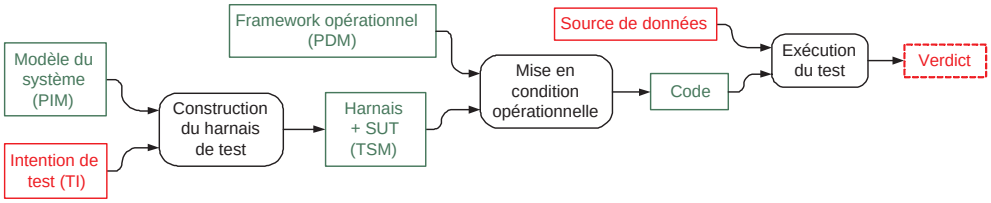


FIGURE 3 – Processus général de test

de cette section. La dernière étape est l'exécution du harnais de test pour déterminer le verdict. La conception de l'oracle et des jeux de données ont une influence à ce niveau opérationnel. Nous y reviendrons dans la section 4.2.

2.3 Construction du harnais de test

Comme le souligne Ghosh [6], il est nécessaire de pouvoir considérer différents contextes pour tester un composant à cause de l'influence des autres composants dans l'assemblage (*race conditions*). Le harnais de test est alors conçu (ou construit) en isolant les composants et services à tester, et en remplaçant tout ou partie de leurs clients (resp. de leurs serveurs) par un *driver* de test (resp. des *mocks*), puis en fournissant les sources de données de test.

Prenons en exemple le test de conformité d'une propriété de sûreté (l'écart entre deux véhicules est suffisant) du service computeSpeed(safeDistance:Integer):Integer du composant mid. Le comportement du service dépend de la distance au précédent, des *positions* et *vitesses* de lui-même et de son prédécesseur. Le *driver* de test (vtd) se charge de placer le service dans son contexte (le composant de droite sur la Figure 4). Ensuite le testeur fixe les besoins du composant mid pour tester le service computeSpeed, en l'occurrence des serveurs pour les services pilotspeed et pilotpos requis par computeSpeed. Deux solutions s'offrent à lui : soit il place des composants de test spécifiques (*mocks*), soit il conserve un peu du contexte applicatif (le composant first). Cette variabilité permet différents types de test dans la même approche et répond à la problématique C3 de Ghosh (cf page 2). Dans la Figure 4 un *mock* spécifique a été conçu car vitesse et position sont liées.

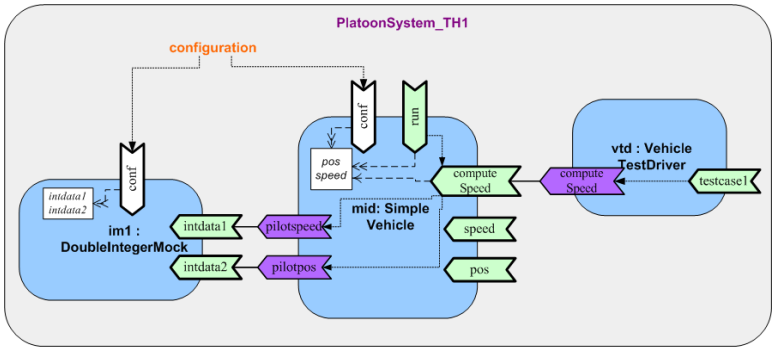


FIGURE 4 – Harnais de test du service computeSpeed du composant mid

Les activités impliquées dans cette partie sont :

- Déterminer le sous-ensemble du PIM concerné. Pour s'assurer de la cohérence et de la pertinence du sous-ensemble choisi, on s'appuie sur les dépendances de services.

- Rechercher et affecter des composants et services de substitution *mocks* pour obtenir une architecture TSM satisfaisant les dépendances des services utilisés.
- Concevoir le driver de test en concrétisant l'intention de test de façon à mettre le service en condition de test : définir la configuration des composants testés, définir la séquence d'appels de service qui place ces composants dans l'état attendu, définir les paramètres à fournir aux services et préciser les données à passer aux *mocks*.

Dans la pratique du test, ces activités, souvent lourdes, relèvent de l'ingénierie de test et sont développées par les testeurs. Par exemple, si computeSpeed prenait ses données sur ses paramètres d'entrée et de sortie, il s'agirait d'un (simple) appel fonctionnel et fournir les données de test serait trivial. Mais ce n'est pas le cas dans notre exemple puisque la distance du prédécesseur, sa vitesse et sa position dépendent de l'état du (composant) véhicule qui précède. De plus la simple configuration des composants ne permet pas ici de fixer ces valeurs. Les données sont fournies grâce à des appels de services : le paramètre safe distance est fourni dans la séquence d'appel du service testcase1 du pilote de test, la position et la vitesse (variables pos et speed de mid) sont fixées par le service de configuration conf de mid, la vitesse et la position du précédent sont fournis par des fonctions abstraites invoquées dans les services initdata1) et (initdata2). Trouver les services à invoquer pour mettre les composants dans un état acceptable pour le test n'est donc pas trivial. Notre objectif est d'assister le testeur durant ces activités, comme nous le détaillerons dans la section 3.

2.4 Mise en condition opérationnelle du harnais

Dans une approche IDM, mettre en condition opérationnelle consiste à produire un modèle adapté à la plateforme cible, notée PDM dans la figure 3. Les premières spécifications sont souvent trop abstraites ou incomplètes pour être exécutables. Chaque service impliqué dans le test doit être exécutable dans un environnement cohérent : ses dépendances doivent être satisfaites par des services compatibles et toutes les opérations utilisées dans l'environnement doivent être assez concrètes ou liées à une opération concrète. Dans notre vision, il s'agit donc de connecter les deux niveaux comme l'illustre la Figure 5 :

- Vérifier que le modèle est exécutable (décrit dans la section 3).
- Déterminer les sources de données concrètes et leur représentation.
- Etablir les *liaisons formelles de données* (*LFD*) c'est-à-dire les points d'entrée du TSM pour les données concrètes.

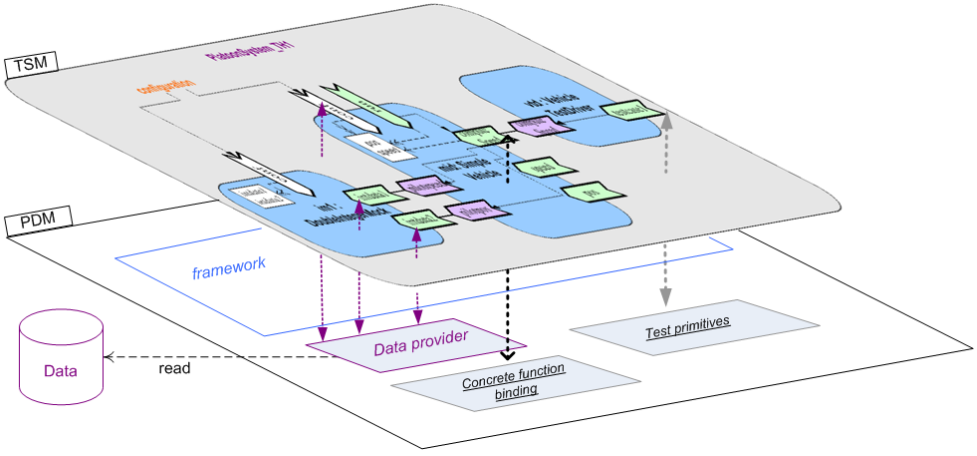


FIGURE 5 – Correspondance entre les données concrètes et les fonctions abstraites

Cette mise en condition opérationnelle présente un coût supplémentaire à cette étape amont de conception. Cependant, les erreurs détectées aussi tôt sont moins coûteuses à corriger que celles de l'implantation des composants et services. En particulier, ces derniers peuvent être implantés sur plusieurs plateformes, multipliant les tests alors qu'une partie de leur comportement est déjà vérifiable pendant la modélisation, comme proposé ici.

3 Assistance à la construction du harnais de test

La seconde proposition de ce travail est un processus général d'aide à la construction d'un harnais de test qui produise un modèle testable et exécutable à partir d'une intention de test plus ou moins détaillée. Ce processus, schématisé dans la Figure 6, couvre la construction du harnais et la partie *exécutabilité* de la mise en condition opérationnelle de la Figure 3. A partir d'un modèle du système (PIM) et d'une intention de test (TI) on construit une application de test exécutable (TSM) et des informations complémentaires (cas de test, transformations).

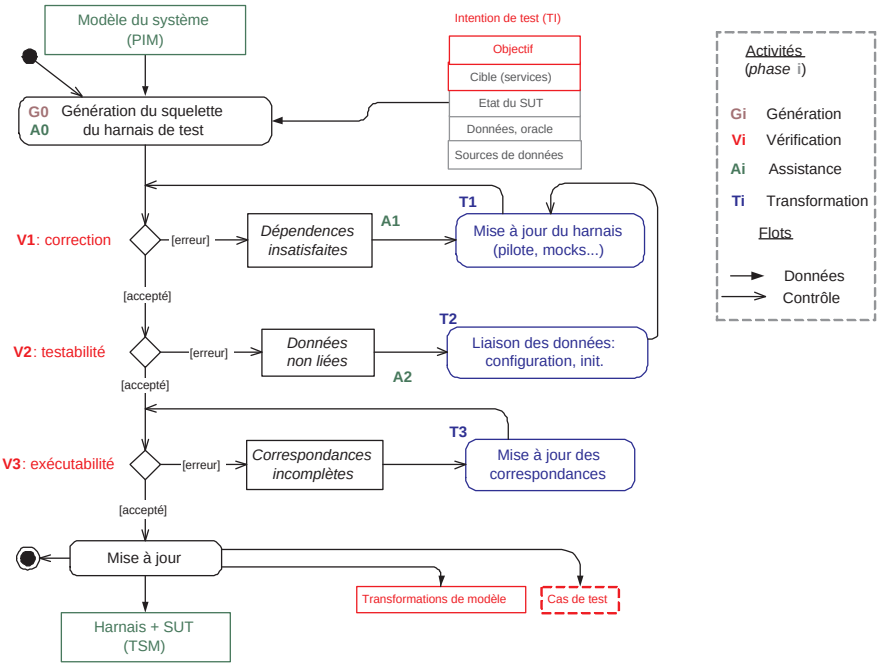


FIGURE 6 – Construction du harnais de test

L'intention de test (introduite en section 2.2) est au départ relativement incomplète, elle dépend finalement de la connaissance qu'a le testeur de l'application. Elle est conçue comme une structure à contenu variable qui par raffinement donnera des cas de test comprenant les sources de données de test (*data binding*), la séquence de test du driver de test, l'expression contextualisée de l'oracle via des services observateurs (*blame assignment*).

Le modèle de l'application évolue lui aussi en parallèle. A mesure que le cas de test se dessine, une nouvelle architecture apparaît, celle de l'application de test. Pour capitaliser ce raffinement, nous souhaitons le représenter comme un processus de transformations. Ainsi, il sera plus facile et plus efficace de faire évoluer le TSM (évolution, non-régression) soit parce que l'intention de test a changé soit parce que le PIM a évolué.

Le processus de construction itère sur des activités de vérification V_i, découverte assistée A_j et transformations T_k. Il se termine lorsque les modèles couvrent le périmètre de test de manière cohérente et complète : l'application est *exécutable*. En particulier, toutes les *données formelles du service DFS* (*formal service data*) qui représentent les données impliquées dans le déroulement d'un service, doivent être couvertes.

A0-G0 Le squelette du harnais est généré par sélection de composants du PIM en fonction de l'intention de test. Les *DFS* sont extraites des services à tester (la source), si le testeur a fourni la cible.

V1 **Correction** La cohérence et la complétude des dépendances de services vis-à-vis de la source sont vérifiées, ainsi que les propriétés de correction mentionnées dans [12].

A1 A partir des signatures des services manquants (et plus si l'interface des services requis est riche) les services candidats sont inférés (soit dans la bibliothèque de composants de tests soit dans le PIM).

T1 On peut générer un pilote à partir des *DFS* et générer l'expression de verdict à partir de l'oracle de l'IT, générer des liens d'assemblages suggérés durant **A1** en ajoutant des substituts proposés (services et composants) ou générés par défaut à partir de sources de données, remplacer des composants et services par des substituts pour réduire le périmètre de test, modifier l'appel du service de configuration des composants, y compris ceux ajoutés dans cette itération, ou attendre la détection d'erreurs.

V2 **Testabilité** Toutes les variables des *DFS* doivent être associées de manière cohérente au harnais. Les paramètres des services apparaissent naturellement dans la *séquence de test*. Les autres variables peuvent se retrouver dans la configuration, la signature des services des substituts ou dans la séquence de test.

A2 Aide à la recherche de données. Pour chaque variable non liée d'un composant, on établit la liste des services de configuration qui peuvent modifier cette variable. Pour chaque donnée externe non liée, une liste des services de substitution est proposée.

T2 Les *LFD* du service sont établies par (i) la configuration des composants en sélectionnant un service de configuration et en lui affectant une source de données, (ii) l'initialisation du test en sélectionnant un service de modification et en lui affectant une source de données, (iii) les liens d'assemblage des services requis responsables de la donnée requise et notamment en le reliant à un substitut.

V3 **Exécutabilité** Les données des composants de test, les substituts, ou la configuration des composants peuvent être spécifiés dans le modèle sous forme de fonctions abstraites de données (signature + assertion). Pour être exécutable, le harnais doit établir des *LFD* de ces fonctions.

T3 Pour chaque correspondance incomplète, on doit affecter une fonction concrète. Les primitives de test (assert) sont automatiquement associées à des fonctions concrètes définies dans une librairie qui contient les fonctions prédéfinies pour les types de base.

L'intérêt de ce processus est méthodologique et il définit un cadre pour la production d'outils d'assistance. Nous abordons ce point dans la section suivante. Certaines étapes sont automatisables, d'autres restent à la charge du testeur, y compris les problématiques non traitées ici, telle que la génération de données de test par exemple.

4 Mise en œuvre et expérimentation

Pour expérimenter les idées présentées dans ce papier nous proposons une mise en œuvre dans l'atelier COSTO support du langage Kmelia. Notre approche est applicable à d'autres modèles à composants tels que SCA, AADL, Sofa ou Fractal, néanmoins ceux-ci permettent principalement d'exprimer l'architecture de composants et nécessitent des spécifications additionnelles pour décrire le comportement des services. Kmelia permet nativement la description à la fois de l'architecture et du comportement, ce qui facilite les

vérifications de cohérence et le prototypage.

4.1 Support au test dans COSTO/Kmelia

Kmelia est un modèle formel et abstrait à composants et services conçu pour développer des systèmes corrects [1]. L'outil COSTO [3], support de Kmelia est un ensemble de plugins Eclipse incluant un éditeur textuel, un contrôleur de type et des outils de vérification de propriétés utilisant principalement des exportations/transformations vers des outils puissants mais plus spécifiques comme MEC, CADP (Lotos), AtelierB ou Rodin (B).

Nous avons tout d'abord développé un *framework* Java (PDM) pour la plateforme cible spécifique (cf Figure 5). Ce *framework* conserve une traçabilité complète des concepts du PIM afin que les diagnostics soient compréhensibles pour le testeur. Concrètement c'est un environnement concurrent dans lequel les services sont des processus tandis que les composants et les canaux sont des moniteurs. Les canaux implantent les liens de clientèle entre services : internes à un composant, entre deux composants assemblés, entre un composant et son composite. Ainsi les communications entre services se font selon une interprétation synchrone avec rendez-vous. Ce *framework* comprend 7 paquetages, plus de 50 classes, 540 méthodes et 3400 lignes de code. Une extension de ce framework permet d'animer la spécification par des actions sur l'évolution des services. Nous avons implanté, au niveau modèle mais aussi du framework, des fonctions prédéfinies pour la vérification de contrats (pre/post conditions, assertions), reprises et enrichies pour établir le verdict des tests (la quantification universelle est toutefois limitée à une variable par expression).

Pour instrumenter le processus de test de la Figure 3, nous avons ensuite implanté différentes activités sous forme de fonctions dans COSTO. Les activités de vérification sont lancées par les menus d'action Eclipse ou réalisées automatiquement après chaque modification de la spécification (vérification de modèle et de détection d'incohérence et d'incomplétude dans les dépendances de service). La vérification d'exécutabilité se fait actuellement lors de génération de code, il faudrait le faire avant. Nous avons aussi développé un plugin de génération du squelette du harnais dans COSTO.

Pour exécuter les tests, nous avons développé un plugin de transformation en code Java, dont les classes héritent du framework d'exécution. Après la génération de code, l'application Java est lancée pour chaque jeu de test, qui rend le verdict du test.

4.2 Analyse des données, oracle et génération des cas de test

Notre approche assiste le testeur pour lui permettre d'exécuter ses tests sur la partie du système qui l'intéresse.

Les choix des composants et services à tester lui reviennent, notre approche permettant aussi bien de considérer chaque composant comme une unité de test que d'intégrer les composants les uns avec les autres pour tester des parties d'assemblage jusqu'au test global du système. Le testeur peut s'appuyer sur des travaux existants pour déterminer un ordre de test des composants et services du système.

Le testeur peut étudier la création des données de test grâce à des travaux exploitant la spécification des composants : travaux de [10] avec des machines à états pour les tests de robustesse, travaux de [15] exploitant les traces (nous verrons à la section 5 que ces travaux exploitent aussi la spécification mais pas pour travailler sur les modèles abstraits des composants). Il ne peut y avoir de test structurel à ce niveau puisque le code de déploiement n'est pas disponible. Dans ce cadre, les tests se basent sur la spécification du service sous test incluant de l'information textuelle et surtout le FSM décrivant le comportement attendu. Le testeur peut exploiter cet automate pour déduire un graphe de flot de contrôle dont il assurera la couverture [3].

3. http://www.lina.sciences.univ-nantes.fr/aelos/wiki/doku.php?id=costo:start

Dans le contexte du test de service, un contrat de qualité de service est généralement disponible. Il peut s'agir du principal oracle des tests. Dans ce cas, il présente l'avantage de pouvoir être utilisé dans plusieurs cas de test. Ce contrat peut ne pas être suffisant pour contrôler complètement la correction des données retournées par un service. Dans ce cas, le testeur pourra exploiter l'analyse de mutation [2] pour évaluer et améliorer la qualité des oracles. Le comportement des services étant modélisé par des FSM nous conseillons de nouveau de ne pas appliquer l'analyse de mutation sur le code mais sur le modèle [13], ce que nous projetons d'étudier prochainement. Par ailleurs, la création d'oracles discrets qui prédisent le résultat attendu et le compare avec le résultat obtenu peut être exploitée (nos premières expérimentations décrites en section 4 utilisent cette méthode).

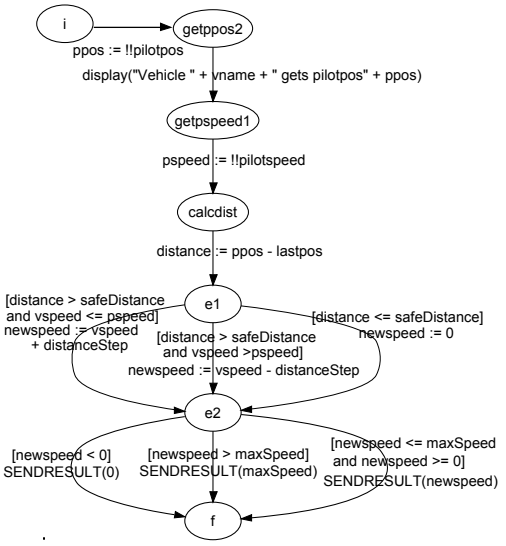
4.3 Application

Tester la correction fonctionnelle de l'application TSM s'opère essentiellement au niveau des services. Nous illustrons dans cette section le test du service computeSpeed du composant mid de l'application platoon introduite dans la section 2 (Figure 2).

L'interface du service computeSpeed est définie dans le Listing 1. Elle comprend deux services requis pour la position et la vitesse du prédécesseur. Le comportement est défini par un système de transitions. La notation !! pilotpos () désigne l'appel du service pilotpos tandis que SendResult est l'envoi du résultat. Ce sont des communications synchrones.

Listing 1 – Service offert computeSpeed

```
provided computeSpeed(safeDistance
   : Integer) : Integer
Interface
   extrequires:
        {pilotpos, pilotspeed}
Pre
     safeDistance >=0
Variables
   distance : Integer;
   newspeed : Integer;
   ppos     : Integer;
   pspeed   : Integer;
Behavior
    #       --> cf figure ci-contre
Post
   Result>=0 && Result<=maxSpeed &&
   ((ppos-lastpos)<safeDistance)
        implies      Result = 0)
End
```



Le Listing 2 précise l'état du véhicule. Les variables d'état vspeed et lastpos correspondent à la vitesse et la dernière position connue ; la variable goalreached signifie que le véhicule a atteint sa destination. L'invariant contraint l'espace des vitesses. L'initialisation fournit les valeurs par défaut.

Listing 2 – Etat de SimpleVehicle

```
VARIABLES
   obs goalreached : Boolean;
   vspeed, lastpos : Integer;
   vname : String;
INVARIANT
   0 <= vspeed and vspeed <= maxSpeed
INITIALIZATION
   goalreached := false;
   vspeed := 0;
   lastpos := 0;
   vname := "Anonymous";
```

Le fonctionnement de computeSpeed dépend des données suivantes : la distance de sécurité entre véhicules safeDistance, la vitesse actuelle vspeed et la position actuelle lastpos et enfin la vitesse et la position du prédécesseur. Ces informations peuvent figurer dans la spécification initiale (voire informelle) du service, ou être extraites par un assistant comme illustré dans la Figure 7(a). Pour compléter l'intention de test, le concepteur de test doit fournir un oracle portant sur ces données et le résultat de l'exécution du service. Dans notre exemple, l'oracle consiste en la comparaison du résultat avec un résultat attendu. La spécification de données concrètes pour un cas de test consiste en un vecteur de valeurs associées aux données : lastpos=10, vspeed=0 pilotpos=10, pilotspeed=80, safeDistance=5, oracledata=45.

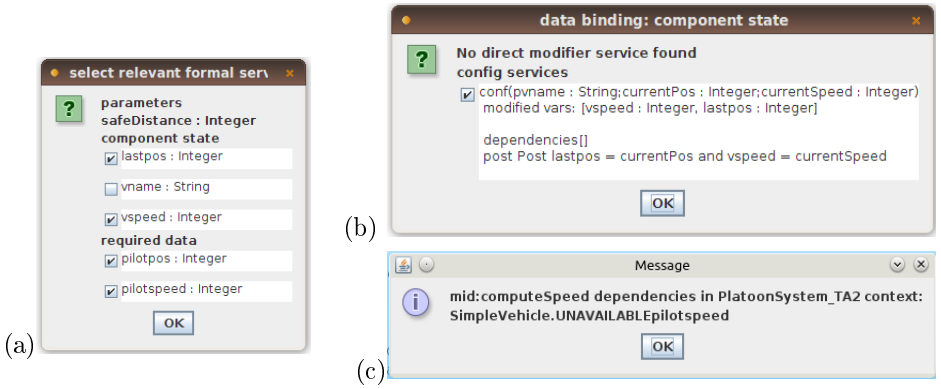


FIGURE 7 – Assistants de conception de harnais

Le processus de construction du harnais n'est pas encore totalement automatisé. Les plugins comportent des fonctions d'assistance au concepteur pour ajouter les dépendances manquantes, calculer les données formelles du service, proposer des sources de données effectives pour les liaisons formelles de données incomplètes (via les bibliothèques prédéfinies de fonctions), trouver des services pour accéder à une partie de l'espace d'état ou modifier une variable. Pour le test du service computeSpeed les étapes du processus sont :

1. La première étape consiste à construire le harnais en sélectionnant le composant SimpleVehicle, la cible de test étant computeSpeed.
2. L'analyse des dépendances de services à partir de computeSpeed met en évidence deux requis pilotpos et pilotspeed. Le processus propose deux alternatives :
   (a) Prendre le composant first dans l'application SUT (Figure 2) et conserver les liens d'assemblage actuels. Dans l'itération suivante il faudra alors satisfaire les requis de first détectés par la vérification V1 (illustrée dans la Figure 7(c)).
   (b) Placer un composant mock. L'assistant propose deux mocks générateurs d'entiers. Nous choisissons de développer notre propre composant mock im1.
3. Pour construire le pilote de test vtd, nous itérons sur la construction de l'environnement :
   (a) L'assistant génère un appel de service sur lequel on associe la donnée safeDistance.
   (b) Il faut ensuite trouver le moyen d'initialiser l'état du composant et associer les données lastpos et vspeed=0. Pour cela, l'assistant (Figure 7(b)) nous liste les services permettant de modifier directement ou indirectement ces variables d'état. Il propose le service d'initialisation `conf`, tout en précisant sa signature, ses éventuelles dépendances et les informations additionnelles le concernant, ici une post-condition.
4. Le calcul de l'oracle est donné par une expression inspirée de la post condition.

5. Le résultat est comparé à la valeur donnée par l'oracle.

La Figure 8 présente la spécification Kmelia de l'architecture du harnais obtenue à la fin du processus, ainsi que le résultat de l'exécution d'un test (ici en console avec des indications d'exécution supplémentaires).

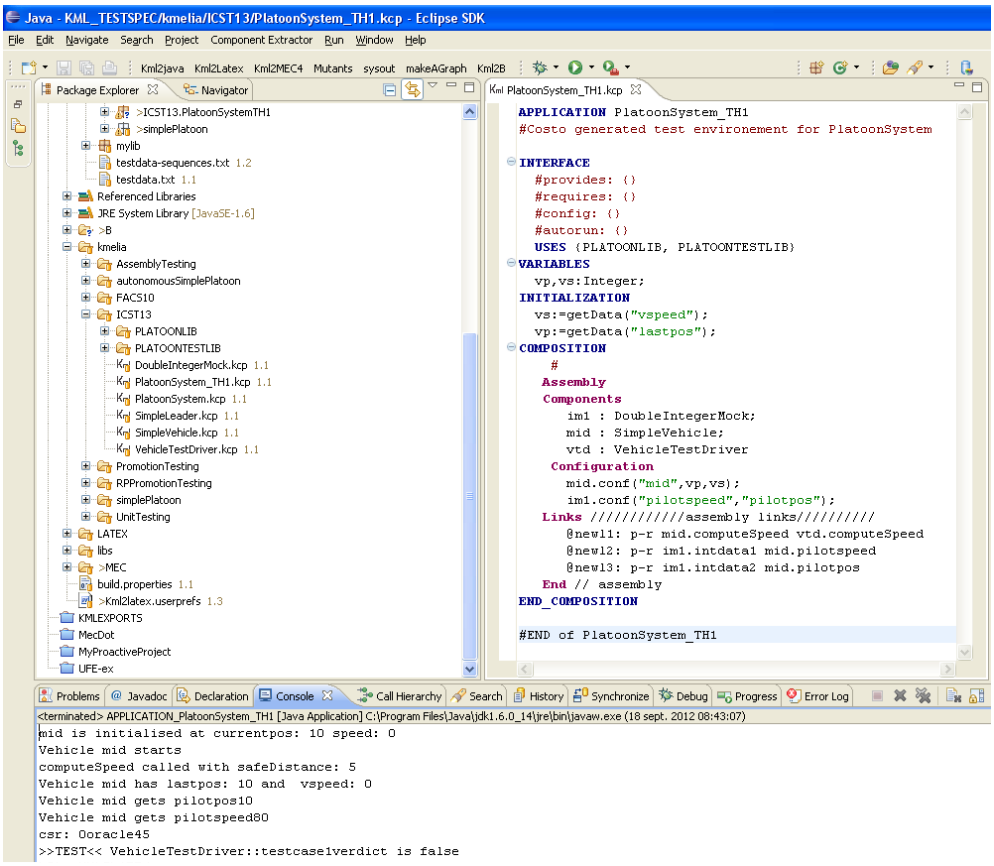


FIGURE 8 – Capture d'écran de l'exécution du test

La clause de configuration contient les appels aux services de configuration du composant mid et du substitut im1. Les variables vp et vs des *données formelles du service* de l'intention de test ont été associées aux variables d'état du composant.

L'intention de test est réalisée par :

1. Un pilote de test muni d'un service contenant la séquence de test et l'oracle sous forme d'un prédicat utilisant la fonction prédéfinie assert (Listing 3).
2. Une bibliothèque de fonctions concrètes (PSM) à associer aux fonctions utilisées dans le modèle (vérification V3). Par exemple, les lignes suivantes établissent la liaison entre la librairie de test Kmelia `mylib` et le code de la plateforme spécifique Java.

   ```
   getData=mylib.PlatoonTestlibMap.getData
   assertT=@WithSelf mylib.PlatoonTestlibMap.assertT
   ```

3. Fichiers textuels de type association pour donner les valeurs de test.

Listing 3 – Service décrivant un cas de test du pilote VehicleTester

```
provided testcase1()
Interface
  extrequires: {computeSpeed}
Variables
  computespeedresult:Integer;
Sequence
{ computespeedresult := !!computeSpeed(getData("safeDistance"));
  //init sequence call
  verdict := (computespeedresult = getData("oracledata") );
  //oracle evaluation
  assertT (verdict);      //transmit verdict
  SendResult()    //end of test service
}
End
```

Pour établir les données de test du service computeSpeed, nous avons utilisé les techniques de couverture du flot de contrôle de l'automate du comportement [3]. Nous avons identifié 9 chemins à couvrir : 4 ne pouvant être atteints par les données, les 5 autres le sont par 3 cas de test chacun. Ces 15 cas de test ont été automatiquement exécutés avec succès.

5 Travaux connexes

En plus des travaux que nous exploitons dans notre approche, d'autres considèrent la génération de tests pour les composants ou à base d'IDM.

Dans [5], Edwards présente une stratégie pour automatiser le test boîte-noire de composant logiciel. Cette stratégie implique une interface riche (avec contrats pre/post). Dans notre approche, nous prenons en plus en compte les communications (protocole). Les composants sont considérés avec une approche orientée-objets, tandis que nous considérons des composants avec une approche orientée-services. Enfin, il se restreint au test unitaire, quand nous permettons en plus de vérifier l'intégration de plusieurs composants.

Dans [10], le test de robustesse de composant est effectué en construisant les tests à partir des machines à états qui décrivent les états valides du système. Néanmoins, l'approche considère les machines à états comme une partie de la spécification et travaille sur le code de déploiement (Java) du composant avec des tests JUnit, alors que nous considérons directement le test du modèle.

Dans [9], Heineman applique une approche de développement dirigée par les test (Test Driven Development) à l'ingénierie des composants logiciels. Les dépendances des composants sont également gérées avec des mocks. Contrairement à notre approche, le niveau de modélisation des composants n'est pas assez expressif pour permettre le test au niveau des modèles. Leur exécution et donc la détection d'erreur ne sont effectuées qu'une fois le code de déploiement disponible.

Dans [17], Zhang introduit la modélisation dirigée par les tests (test-driven modeling) pour appliquer une approche XP à un processus de développement dirigé par les modèles. L'approche permet de concevoir les tests avant la modélisation (alors que nous construisons nos tests en les basant sur le modèle) mais à nouveau ici les tests sont appliqués sur la plateforme de déploiement (dès que la description comportementale est suffisante pour générer un code spécifique d'exécution des tests).

Dans [14], Rocha et al. étudient la génération de cas de test et de *stubs* à partir de diagramme d'activité UML. Cette modélisation ne fournit pas les informations de dépendances de services et de données qui nous ont permis d'inférer sur la construction du harnais de test. De plus ces *stubs* permettent moins de vérifications que les *mocks* que nous proposons. Leur méthode manque aussi de support d'assistance.

Par ailleurs, dans [15], les auteurs adaptent le test de composants selon leur utilisation dans un système. Ils exploitent aussi la définition par LTS du comportement des composants, mais ne travaille pas niveau modèle. Notre framework permet aussi d'adapter le test par la définition d'un périmètre de test et la sélection de mock. Il s'agit d'un travail que nous pouvons étudier dans le futur pour intégrer une génération automatique de données de test à notre approche.

6 Conclusion

Dans cet article, nous avons décrit une approche pragmatique du test de modèles à composants et services, qui accompagne la vérification formelle de la syntaxe et de propriétés fonctionnelles en intégrant la détection précoce d'erreurs dans un processus de développement dirigé par les modèles. Les tests sont réalisés au niveau modèle, apportant une abstraction nécessaire au découplage applicatif et technique, mais facilitant aussi l'évolution des tests en cohérence avec celle des modèles, qui est fréquente dans les phases initiales de conception.

L'abstraction réduit la complexité du travail de conception des tests mais reste une activité lourde. Nous proposons donc un processus guidé et des outils pour aider le concepteur de test de modèles à composant dans la construction de harnais de test. Cela vise à rapprocher le modèle du système et le modèle de test et permettre leur exécution. L'aide consiste en la détection d'incohérence ou d'incomplétude dans ce rapprochement et dans la proposition de solutions. Le modèle du harnais est produit par un ensemble de transformations de modèles, favorisant ainsi la réutilisation, l'adaptation, et l'évolution des tests.

La proposition est expérimentée avec Kmelia, un langage permettant de modéliser les composants et services, y compris les composants de test. COSTO permet de vérifier et d'exécuter les modèles Kmelia sur la plateforme d'exécution cible spécifique qui conserve les concepts abstraits dans l'application générée (forte traçabilité). Les outils d'assistance implantés dans COSTO permettent de construire, à partir de tout ou partie d'une application, le harnais de test. Les outils de détection d'erreurs et d'exécution sont totalement implantés, ceux qui proposent des corrections le sont partiellement. Le processus complet n'est pas encore implanté mais des expérimentations sont menées pour réaliser la construction avec un ensemble de transformations de modèles en ATL.

Les travaux en cours portent sur l'implantation du processus et l'enchaînement des cas de test mais aussi sur le support à la génération automatique de composants et services (pilote de test, bouchons) et l'adaptation de composants existants à un contexte donné.

Concernant la description des tests, nous projetons d'exploiter plus finement la description des contrats de services, issus du modèle, pour affiner la construction de l'oracle et déterminer les logiques les plus adaptées à leur vérification. Par exemple, la cohérence des contrats fonctionnels dans Kmelia est vérifiée à l'aide des installations extérieures de COSTO [12] afin qu'ils fournissent une source fiable d'oracles. Les propriétés de traces et les invariants dynamiques pourraient être exprimés dans le modèle et passés à travers des versions personnalisées de la fonction intégrée assert. Ils nécessiteront d'autres observateurs simples à mettre en œuvre dans notre génération de code, mais plus difficiles à rendre indépendants des plate-formes en préservant l'encapsulation.

Nous projetons de comparer notre approche avec xUnit. En particulier, les étapes que nous proposons pour créer le harnais de test peuvent être comparés au setUp initialisant des tests unitaires. Un tearDown n'est pas encore implanté dans notre approche, ce qui peut nécessiter une nouvelle création du harnais pour assurer l'indépendance de tests.

Afin d'appliquer l'analyse de mutation et obtenir des informations supplémentaires avec exactitude sur les données de test et les contrats d'oracle, nous allons également étudier des modèles de fautes pour la spécification de comportement. Nous appliquerons les mutations sur les modèles plutôt que sur le code généré, car une telle approche pourrait être intégrée

dans l'ingénierie des modèles. De la même manière, nous projetons d'étudier l'intégration de nos tests niveau modèle dans le développement dirigé par les modèles, comme proposé dans [4]. Cela permettrait par transformation de générer des tests spécifiques aux différentes plateformes d'implantation des composants.

Références

[1] P. André, G. Ardourel, C. Attiogbé, and A. Lanoix. Using assertions to enhance the correctness of kmelia components and their assemblies. *ENTCS*, 263 :5 – 30, 2010. Proceedings of FACS 2009.

[2] B. Baudry, Y. Le Traon, and H. Vu Le. Testing-for-trust : the genetic selection model applied to component qualification. In *Proceedings of TOOLS Europe*, volume 33, pages 108–119. IEEE Computer Society, June 2000.

[3] R. V. Binder. *Testing object-oriented systems : models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[4] M. Born, I. Schieferdecker, H.-g. Gross, and P. Santos. Model-driven development and testing - a case study. In *First European Workshop on MDA with Emphasis on Industrial Application*, pages 97–104. Twente Univ., 2004.

[5] S. H. Edwards. A Framework for Practical, Automated Black-Box Testing of Component-Based Software. *Software Testing, Verification and Reliability*, 11(2) :97–111, 2001.

[6] S. Ghosh and A. P. Mathur. Issues in testing distributed component-based systems. In *In First International ICSE Workshop on Testing Distributed Component-Based Systems*, 1999.

[7] M. Gogolla, J. Bohling, and M. Richters. Validating uml and ocl models in use by automatic snapshot generation. *Software and Systems Modeling*, 4(4) :386–398, 2005.

[8] H.-G. Gross. *Component-based Software Testing With Uml*. SpringerVerlag, 2004.

[9] G. Heineman. Unit testing of software components with inter-component dependencies. In *Component-Based Software Engineering*, volume 5582 of *LNCS*, pages 262–273. Springer Berlin / Heidelberg, 2009.

[10] B. Lei, Z. Liu, C. Morisset, and X. Li. State based robustness testing for components. *Electr. Notes Theor. Comput. Sci.*, 260 :173–188, 2010.

[11] J. Marino and M. Rowley. *Understanding SCA (Service Component Architecture)*. Addison-Wesley Professional, 1st edition, 2009.

[12] M. Messabihi, P. André, and C. Attiogbé. Multilevel contracts for trusted components. In J. Cámara, C. Canal, and G. Salaün, editors, *WCSI*, volume 37 of *EPTCS*, pages 71–85, 2010.

[13] S. Pinto Ferraz Fabbri, M. Delamaro, J. Maldonado, and P. Masiero. Mutation analysis testing for finite state machines. In *Proceedings., 5th International Symposium on Software Reliability Engineering*, pages 220 –229, nov 1994.

[14] C. R. Rocha and E. Martins. A method for model based test harness generation for component testing. *Journal of the Brazilian Computer Society*, 14 :7 – 23, 03 2008.

[15] B. Schätz and C. Pfaller. Integrating component tests to system tests. *Electr. Notes Theor. Comput. Sci.*, 260 :225–241, 2010.

[16] G. Shanks, E. Tansley, and R. Weber. Using ontology to validate conceptual models. *Commun. ACM*, 46(10) :85–89, Oct. 2003.

[17] Y. Zhang. Test-driven modeling for model-driven development. *IEEE Software*, 21(5) :80–86, 2004.